InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy

Mengjia Yan†, Jiho Choi†, Dimitrios Skarlatos, Adam Morrison*, Christopher W. Fletcher, and Josep Torrellas
University of Illinois at Urbana-Champaign *Tel Aviv University
{myan8, jchoi42, skarlat2}@illinois.edu, mad@cs.tau.ac.il, {cwfletch, torrella}@illinois.edu
†Authors contributed equally to this work.

***Abstract*— Hardware speculation offers a major surface for micro-architectural covert and side channel attacks. Unfortunately, defending against speculative execution attacks is challenging. The reason is that speculations destined to be squashed execute incorrect instructions, outside the scope of what programmers and compilers reason about. Further, any change to micro-architectural state made by speculative execution can leak information.**

In this paper, we propose *InvisiSpec*, a novel strategy to defend against hardware speculation attacks in multiprocessors by making speculation invisible in the data cache hierarchy. InvisiSpec blocks micro-architectural covert and side channels through the multiprocessor data cache hierarchy due to speculative loads. In InvisiSpec, unsafe speculative loads read data into a speculative buffer, without modifying the cache hierarchy. When the loads become safe, InvisiSpec makes them visible to the rest of the system. InvisiSpec identifies loads that might have violated memory consistency and, at this time, forces them to perform a validation step. We propose two InvisiSpec designs: one to defend against Spectre-like attacks and another to defend against futuristic attacks, where any speculative load may pose a threat. Our simulations with 23 SPEC and 10 PARSEC workloads show that InvisiSpec is effective. Under TSO, using fences to defend against Spectre attacks slows down execution by 74% relative to a conventional, insecure processor; InvisiSpec reduces the execution slowdown to only 21%. Using fences to defend against futuristic attacks slows down execution by 208%; InvisiSpec reduces the slowdown to 72%.

I. Introduction

The recent disclosure of Spectre [1] and Meltdown [2] has opened a new chapter in hardware security pertaining to the dangers of speculative execution. Hardware speculation can cause execution to proceed in ways that were not intended by the programmer or compiler. Until recently, this was not thought to have security implications, as incorrect speculation is guaranteed to be squashed by the hardware. However, these attacks demonstrate that squashing incorrect speculative paths is insufficient for security.

Spectre and Meltdown monitor the micro-architectural footprint left by speculation, such as the state left by wrong-path speculative loads in the cache. This footprint enables micro-architectural covert or side channels, where an adversary can infer the speculative thread's actions. In Meltdown, a user-level attacker is able to execute a privileged memory load and leak the memory contents using a cache-based covert channel, all before the illegal memory load triggers an exception. In Spectre, the attacker poisons the branch predictor or branch target buffer to force the victim's code to speculate down specific paths of the attacker's choosing. Different types of attacks are possible, such as the victim code reading arbitrary memory locations by speculating that an array bounds check succeeds—allowing the attacker to glean information through a cache-based side channel, as shown in Figure 1. In this case, unlike in Meltdown, there is no exception-inducing load.

```
1  // cache line size is 64 bytes
2  // secret value V is 1 byte
3  // victim code begins here:
4  uint8 A[10];
5  uint8 B[256*64];
6  // B size: possible V values * line size
7  void victim(size_t a) {
8    if (a < 10)
9      junk = B[64 * A[a]];
10 }
11 // attacker code begins here:
12 train(); // train if condition to be true
13 flush(B); // flush every cache line of B
14 call victim(X - &A);
15 scan(B); // access each cache line of B
```

Fig. 1: Spectre variant 1 attack example, where the attacker obtains secret value V stored at address X. If the *If* condition is predicted as true, then the V-th cache line in B is loaded to the cache even though the load is eventually squashed.

It can be argued that Meltdown can be fixed with a relatively modest implementation change: mark the data loaded by an exception-triggering load as unreadable. However, defending against Spectre and other forms of speculative execution attacks that do not cause exceptions presents a two-fold challenge. First, potentially *any* instruction performing speculation can result in an attack. The reason is that speculations destined to be squashed inherently execute incorrect instructions (so-called *transient* instructions), outside the scope of what programmers and compilers reason about. Since program behavior is undefined, speculative execution attacks will be able to manifest in many ways, similar to attacks exploiting lack of memory safety [3]. Second, speculative code that makes *any* change to micro-architectural state—e.g., cache occupancy [4] or cache coherence [5] state—can create covert or side channels.

Given these issues, it is unsurprising that current defenses against Spectre suffer from either high performance overheads or incomplete security. For example, consider Spectre variant 1, where mis-predictions on direct conditional branches can lead to attacks such as array bounds check bypassing [1]. There is no current hardware proposal to block it, except for disabling all speculation, which is clearly unacceptable. In software, current mitigations are to place LFENCE instructions after sensitive branches, replace branches with CMOVs, and mask index bits before the array look-up [6]. Unfortunately, each of these has usage and/or performance deficiencies—e.g., fences block good speculation, and masking only helps prevent bounds-bypass

attacks. In addition, these techniques only attempt to block speculation on branches.

In this paper, we propose *InvisiSpec*, a novel strategy to defend against hardware speculation attacks in multiprocessors, by making speculation invisible in the data cache hierarchy. The goal is to block micro-architectural covert and side channels through the multiprocessor data cache hierarchy due to speculative loads—e.g., channels stemming from cache set occupancy, line replacement information, and cache coherence state. We wish to protect against not only Spectre-like attacks based on branch speculation; we also want to protect against futuristic attacks where any speculative load may pose a threat.

InvisiSpec's micro-architecture is based on two mechanisms. First, unsafe speculative loads read data into a new *Speculative Buffer* (SB) instead of into the caches, without modifying the cache hierarchy. Data in the SB do not observe cache coherence transactions, which may result in missing memory consistency violations. Our second mechanism addresses this problem. When a speculative load is finally safe, the InvisiSpec hardware makes it visible to the rest of the system by reissuing it to the memory system and loading the data into the caches. In this process, if InvisiSpec considers that the load could have violated memory consistency, InvisiSpec *validates* that the data that the load first read is correct—squashing the load if the data is not correct.

To summarize, this paper makes the following contributions:

1) We present a generalization of speculative execution attacks that goes beyond current Spectre attacks, where any speculative load may pose a threat.
2) We present a micro-architecture that blocks side and covert channels during speculation in a multiprocessor data cache hierarchy.
3) We simulate our design on 23 SPEC and 10 PARSEC workloads. Under TSO, using fences to defend against Spectre attacks slows down execution by 74% relative to a conventional, insecure processor; InvisiSpec reduces the execution slowdown to only 21%. Using fences to defend against futuristic attacks slows down execution by 208%; InvisiSpec reduces the slowdown to 72%.

II. Background and Terminology

A. Out-of-order and Speculative Execution

Out-of-order execution. Dynamically-scheduled processors [7] execute data-independent instructions in parallel, out of program order, and thereby exploit instruction-level parallelism [8] to improve performance. Instructions are *issued* (enter the scheduling system) in program order, *complete* (execute and produce their results) possibly out of program order, and finally *retire* (irrevocably modify the architected system state) in program order. In-order retirement is implemented by queueing instructions in program order in a reorder buffer (ROB) [9], and removing a completed instruction from the ROB only once it reaches the ROB head, i.e., after all prior instructions have retired.

Speculative execution. Speculative execution is the execution of an instruction before its validity can be made certain. If, later, the instruction turns out to be valid, it is eventually retired, and its speculative execution has improved performance. Otherwise, the instruction will be squashed and the processor's state rolled back to the state before the instruction. (At the same time, all of the subsequent instructions in the ROB also get squashed.) Strictly speaking, an instruction executes speculatively in an out-of-order processor if it executes before it reaches the head of the ROB.

There are multiple reasons why a speculatively-executed instruction may end up being squashed. One reason is that a preceding branch resolves and turns out to be mispredicted. Another reason is that, when a store or a load resolves the address that it will access, the address matches that of a later load that has already obtained its data from that address. Another reason is memory consistency violations. Finally, other reasons include various exceptions and the actual squash of earlier instructions in the ROB.

B. Memory Consistency & Its Connection to Speculation

A memory consistency model (or memory model) specifies the order in which memory operations are performed by a core, and are observed by other cores, in a shared-memory system [10]. When a store retires, its data is deposited into the write buffer. From there, when the memory consistency model allows, the data is merged into the cache. We say that the store is *performed* when the data is merged into the cache, and becomes observable by all the other cores. Loads can read from memory before they reach the ROB head. We say that the load is *performed* when it receives data. Loads can read from memory and be performed out of order—i.e., before earlier loads and stores in the ROB are. Out-of-order loads can lead to memory consistency violations, which the core recovers from using the squash and rollback mechanism of speculative execution [11]. The details depend on the memory model; we describe two common models below, which we assume as the baseline to which InvisiSpec is added.

Total Store Order (TSO) [12], [13] is the memory model of the x86 architecture. TSO forbids all observable load and store reorderings except store→load reordering, which is when a load bypasses an earlier store to a different address. Implementations prevent observable load→load reordering by ensuring that the value a load reads when it is performed remains valid when the load retires. This guarantee is maintained by squashing a load that has performed, but not yet retired, if the core receives a cache invalidation request for (or suffers a cache eviction of) the line read by the load. Store→store reordering is prevented by using a FIFO write buffer, ensuring that stores perform in program order. If desired, store→load reordering can be prevented by separating the store and the load with a fence instruction, which does not complete until all prior accesses are performed. Atomic instructions have fence semantics.

Release Consistency (RC) [14] allows any reordering, except across synchronization instructions. Loads and stores may not be reordered with a prior acquire or with a subsequent release. Therefore, RC implementations squash performed loads upon receiving an invalidation of their cache line only if there is a prior non-retired acquire, and have a non-FIFO write buffer.

III. Threat Model

Speculative execution attacks exploit the side effects of *transient* instructions—i.e., speculatively-executed instructions that are destined to be squashed. Specifically, *victim* programs speculatively execute *access* instructions that can write secrets into general purpose registers or the store queue (see Section IV for examples). Before such access instructions are squashed, further speculative *transmit* instructions may execute and leak those secrets to adversarial code, the *receiver*, over a covert channel. As the access and transmit instructions are transient, they would not have executed in a non-speculative execution.

Since predictors are not perfect, dangerous access and transmit instructions may execute in the normal course of execution. We make conservative (but realistic [1]) assumptions, and allow the adversary to run arbitrary code before and during victim execution to influence the victim program's speculation—e.g., by changing branch predictor state. We assume that the processor hardware and the operating system function correctly.

A. Cache Hierarchy-Based Side Channels

In almost all of the known speculative execution attacks, the adversary transmits over a cache-based side channel. The adversary exploits the fact that transient load instructions modify the cache state, which allows the receiver to learn information out of the victim's cache state changes.[1] The two most popular techniques for cache-based side channel attacks are Prime+Probe (where the receiver monitors conflicts in the cache through its own evictions) and Flush+Reload (where the receiver uses instructions such as x86's `clflush` to evict the victim's cache lines). Cache-based side channels have been demonstrated within and across cores [16], and on both inclusive and non-inclusive cache hierarchies [17]. Recent work has also shown side channels from cache coherence state changes [5] and data TLB state [18].

B. Channels Protected

InvisiSpec blocks covert and side channels that are constructed by monitoring the state of any level of a multiprocessor data cache hierarchy. This includes cache state such as occupancy (e.g., what lines live in what sets), replacement information, and coherence information (e.g., whether a line is in the Exclusive or Shared state). It also includes data TLB state (e.g., what entries live in the TLB). Our goal is to provide a complete defense for the data cache hierarchy,[2] as it has been proven to lead to practical attacks [1], [2]. Note that our mechanism does not preclude protecting additional channels.

Out of scope. Physical side channels (e.g., power [19] or EM [20]) are out of scope. We also do not protect micro-architectural channels other than those listed above. One such example is monitoring the timing of execution units [1], including floating-point units [21] and SIMD units [15]—which can be mitigated by not scheduling the victim and adversary in adjacent SMT contexts. Other examples are contention on the network-on-chip [22] or DRAM [23], which may allow an adversary to learn coarse-grain information about the victim. We leave protecting these channels as future work.

[1]The exception is NetSpectre [15]. It uses a timing channel created by speculative execution, which powers-up SIMD units and thus speeds-up later SIMD instructions.

[2]For simplicity, we do not describe the protection of the instruction cache or instruction TLB. However, we believe they can be protected with similar or even simpler structures as those we propose for the data cache hierarchy.

C. Settings

We block the side and covert channels described in Section III-B in the following settings, which differ in where the adversarial receiver code runs relative to the victim program.

SameThread. The receiver runs in the same thread as the victim. This case models scenarios such as JavaScript, where adversarial code, e.g., a malicious advertisement or tab, may be JITed into an otherwise honest program. As such, we assume that malicious and honest code temporally multiplex the core. They share the private caches and TLBs, and the attacker can snapshot the state of these resources when it becomes active. On the other hand, it is not possible for malicious code to learn fine-grain behavior such as exactly when an execution unit is used.

CrossCore. The receiver runs on a separate physical core. This case models a multi-tenant cloud scenario where different VMs are scheduled to different physical cores on the same socket. The adversary can monitor the last-level cache and its directory—e.g., how line coherence state changes over time.

SMT. The receiver runs in an adjacent SMT context of the same core. The threads may share the private caches and/or the TLB. We note that while our security objective in Section III-B is only to protect against the monitoring of cache and TLB state, SMT presents a broader attack surface: fine-grain structures at the core such as execution units can also be used as covert or side channels. We leave blocking such channels as future work.

IV. Understanding Speculative Execution Attacks

We discuss the events that can be the source of transient instructions leading to a security attack. Table I shows examples of such attacks and what event creates the transient instructions.

Attack	What Creates the Transient Instructions
Meltdown	Virtual memory exception
L1 Terminal Fault	
Lazy Floating Point	Exception reading a disabled or privileged register
Rogue System Register Read	
Spectre	Control-flow misprediction
Speculative Store Bypass	Address alias between a load and an earlier store
Futuristic	Various events, such as: • Exceptions • Control-flow mispredictions • Address alias between a load and an earlier store • Address alias between two loads • Memory consistency model violations • Interrupts

TABLE I: Understanding speculative execution attacks.

Exceptions. Several attacks exploit speculation past an exception-raising instruction. The processor squashes the

execution when the exception-raising instruction reaches the head of the ROB, but by this time, dependent transmitting instructions can leak data. The Meltdown [2] and L1 Terminal Fault (L1TF) [24], [25] attacks exploit virtual memory-related exceptions. Meltdown reads a kernel address mapped as inaccessible to user space in the page table. L1TF reads a virtual address whose page table entry (PTE) marks the physical page as not present, but the physical address is still loaded.

The Lazy Floating Point (FP) State Restore attack [26] reads an FP register after the OS has disabled FP operations, thereby reading the FP state of another process. The Rogue System Register Read [26] reads privileged system registers.

Control-flow misprediction. Spectre attacks [1] exploit control-flow speculation to load from an arbitrary memory address and leak its value. Variant 1 performs an out-of-bounds array read, exploiting a branch misprediction of the array bounds check [1]. Other variants direct the control flow to an instruction sequence that leaks arbitrary memory, either through indirect branch misprediction [1], return address misprediction [27], or an out-of-bounds array write that redirects the control flow (e.g., by overwriting a return address on the stack) [28].

Memory-dependence speculation. Speculative Store Bypass (SSB) attacks [29] exploit a speculative load that bypasses an earlier store whose address is unresolved, but will resolve to alias the load's address [30]. Before the load is squashed, it obtains stale data, which can be leaked through subsequent dependent instructions. For example, suppose that a JavaScript runtime or virtual machine zeroes out memory allocated to a user thread. Then, this attack can be used to peek at the prior contents of an allocated buffer. We discovered the SSB attack in the process of performing this work. While our paper was under review, a security analysis independently conducted by Microsoft and Google found this vulnerability [29].

A comprehensive model of speculative execution attacks. Transient instructions can be created by many events. As long as an instruction can be squashed, it can create transient instructions, which can be used to mount a speculative execution attack. Since we focus on cache-based covert and side channel attacks, we limit ourselves to load instructions.

We define a *Futuristic* attack as one that can exploit any speculative load. Table I shows examples of what can create transient instructions in a Futuristic attack: all types of exceptions, control-flow mispredictions, address alias between a load and an earlier store, address alias between two loads, memory consistency model violations, and even interrupts.

In the rest of the paper, we present two versions of InvisiSpec: one that defends against Spectre attacks, and one that defends against Futuristic attacks.

V. InvisiSpec: Thwarting Speculation Attacks

A. Main Ideas

1) Unsafe Speculative Loads: Strictly speaking, any load that initiates a read before it reaches the head of the ROB is a speculative load. In this work, we are interested in the (large) subset of speculative loads that can create a security vulnerability due to speculation. We call these loads *Unsafe Speculative Loads (USLs)*. The set of speculative loads that are USLs depends on the attack model.

In the Spectre attack model, USLs are the speculative loads that follow an unresolved control-flow instruction. As soon as the control-flow instruction resolves, the USLs that follow it in the correct path of the branch transition to *safe loads*—although they remain speculative.

In our Futuristic attack model, USLs are all the *speculative loads that can be squashed by an earlier instruction*. A USL transitions to a *safe load* as soon as it becomes either (i) non-speculative because it reaches the head of the ROB or (ii) speculative non-squashable by any earlier instruction (or *speculative non-squashable* for short). *Speculative non-squashable* loads are loads that both (i) are not at the head of the ROB and (ii) are preceded in the ROB only by instructions that cannot be squashed by any of the squashing events in Table I for Futuristic. Note that in the table, one of the squashing events is interrupts. Hence, making a load safe includes delaying interrupts until the load reaches the head of the ROB.

To understand why these two conditions allow a USL to transition to a safe load consider the following. A load at the ROB head cannot itself be transient; it can be squashed (e.g., due to an exception), but it is a correct instruction. The same can be said about speculative non-squashable loads. This is because, while not at the ROB head, they cannot be squashed by any earlier instruction and, hence, can be considered a logical extension of the instruction at the ROB head.

2) Making USLs Invisible: The idea behind InvisiSpec is to make USLs *invisible*. This means that a USL cannot modify the cache hierarchy in any way that is visible to other threads, including the coherence states. A USL loads the data into a special buffer that we call *Speculative Buffer* (SB), and not into the local caches. As indicated above, there is a point in time when the USL can transition to a safe load. At this point, called the *Visibility Point*, InvisiSpec takes an action that will make the USL *visible*—i.e., will make all the side effects of the USL in the memory hierarchy apparent to all other threads. InvisiSpec makes the USL visible by re-loading the data, this time storing the data in the local caches and changing the cache hierarchy states. The load may remain speculative.

3) Maintaining Memory Consistency: The *Window of Suppressed Visibility* for a load is the time period between when the load is issued as a USL and when it makes itself visible. During this period, since a USL does not change any coherence state, the core may fail to receive invalidations directed to the line loaded by the USL. Therefore, violations of the memory consistency model by the load run the risk of going undetected. This is because such violations are ordinarily detected via incoming invalidations, and are solved by squashing the load. To solve this problem, InvisiSpec may have to perform a *validation* step when it re-loads the data at the load's visibility point.

It is possible that the line requested by the USL was already in the core's L1 cache when the USL was issued and the

line loaded into the SB. In this case, the core may receive an invalidation for the line. However, the USL ignores such invalidation, as the USL is invisible, and any effects of this invalidation will be handled at the visibility point.

4) Validation or Exposure of a USL: The operation of reloading the data at the visibility point has two flavors: *Validation* and *Exposure*. Validation is the way to make visible a USL that would have been squashed during the Window of Suppressed Visibility (due to memory consistency considerations) if, during that window, the core had received an invalidation for the line loaded by the USL. A validation operation includes comparing the actual bytes used by the USL (as stored in the SB) to their most up-to-date value being loaded from the cache hierarchy. If they are not the same, the USL and all its successive instructions are squashed. This is required to satisfy the memory consistency model. This step is reminiscent of Cain and Lipasti's value-based memory ordering [31].

Validations can be expensive. A USL enduring a validation cannot retire until the transaction finishes—i.e., the line obtained from the cache hierarchy is loaded into the cache, the line's data is compared to the subset of it used in the SB, and a decision regarding squashing is made. Hence, if the USL is at the ROB head and the ROB is full, the pipeline stalls.

Thankfully, InvisiSpec identifies many USLs that could not have violated the memory consistency model during their Window of Suppressed Visibility, and allows them to become visible with a cheap *Exposure*. These are USLs that would *not* have been squashed by the memory consistency model during the Window of Suppressed Visibility if, during that window, the core had received an invalidation for the line loaded by the USL. In an exposure, the line returned by the cache hierarchy is simply stored in the caches without comparison. A USL enduring an exposure can retire as soon as the line request is sent to the cache hierarchy. Hence, the USL does not stall the pipeline.

To summarize, there are two ways to make a USL visible: validation and exposure. The memory consistency model determines which one is needed. Figure 2 shows the timeline of a USL with validation and with exposure.

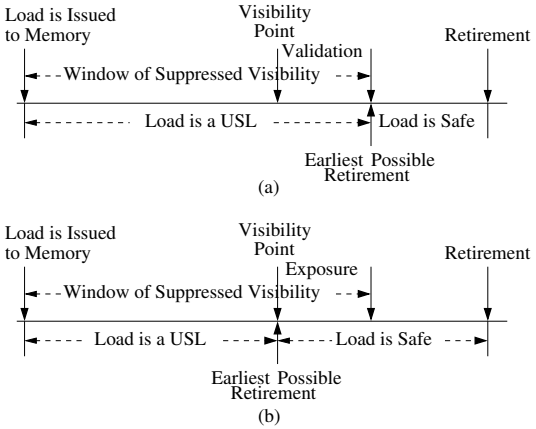

Fig. 2: Timeline of a USL with validation (a) and exposure (b).

B. InvisiSpec Operation

A load in InvisiSpec has two steps. First, when it is issued to memory as a USL, it accesses the cache hierarchy and obtains the current version of the requested cache line. The line is only stored in the local SB, which is as close to the core as the L1 cache. USLs do not modify the cache coherence states, cache replacement algorithm states, or any other cache hierarchy state. No other thread, local or remote, can see any changes. However, the core uses the data returned by the USL to make progress. The SB stores lines rather than individual words to exploit spatial locality.

When the USL can be made visible, and always after it has received its requested cache line, the hardware triggers a validation or an exposure transaction. Such a transaction re-requests the line again, this time modifying the cache hierarchy, and bringing the line to the local caches. As detailed in Section V-A4, validation and exposure transactions operate differently and have different performance implications.

We consider two attack models, Spectre and Futuristic, and propose slightly different InvisiSpec designs to defend against each of these attacks. In our defense against the Spectre attack, a USL reaches its visibility point when all of its prior control-flow instructions resolve. At that point, the hardware issues a validation or an exposure transaction for the load depending on the memory consistency model and the load's position in the ROB (Section V-C). If multiple USLs can issue validation or exposure transactions, the transactions have to start in program order, but can otherwise all overlap (Section V-D).

In our defense against Futuristic attacks, a USL reaches its visibility point only when: (i) it is not speculative anymore because it is at the head of the ROB, or (ii) it is still speculative but cannot be squashed anymore. At that point, the hardware issues a validation or an exposure for the load depending on the memory consistency model and the load's position in the ROB (Section V-C). If multiple USLs can issue validation or exposure transactions, the transactions have to be issued in program order. However, when a validation transaction is issued, no subsequent validation or exposure transaction can overlap with it; they all have to wait to be issued until the validation is complete (Section V-D). On the other hand, when an exposure transaction is issued, all subsequent exposure transactions up to, and including, the next validation transaction can overlap with it (Section V-D).

Overall, in the Spectre and Futuristic defense designs, the only pipeline stalls may occur when a validation transaction holds up the retirement of a load at the head of the ROB and the ROB is full. This is more likely in Futuristic than in Spectre.

We call these designs *InvisiSpec-Spectre* (or *IS-Spectre*) and *InvisiSpec-Future* (or *IS-Future*). They are shown in the first row of Table II. For comparison, the second row shows how to defend against these same attacks using fence-based approaches—following current proposals to defend against Spectre [32]. We call these designs *Fence-Spectre* and *Fence-Future*. The former places a fence after every indirect or conditional branch; the latter places a fence before every load.

	Defense Against Spectre	Defense Against Futuristic
Invisi-Spec Based	*InvisiSpec-Spectre*: 1) Perform invisible loads in the shadow of an unresolved BR; 2) Validate or expose the loads when the BR resolves	*InvisiSpec-Future*: 1) Perform an invisible load if the load can be squashed while speculative; 2) Validate or expose the load when it becomes either (i) non-speculative or (ii) un-squashable speculative
Fence Based	*Fence-Spectre*: Place a fence after every BR	*Fence-Future*: Place a fence before every load

TABLE II: Summary of designs (BR means indirect or conditional branch.)

Compared to the fence-based designs, InvisiSpec improves performance. Specifically, loads are speculatively executed as early as in conventional, insecure machines. One concern is that validation transactions for loads at the head of the ROB may stall the pipeline. However, we will show how to maximize the number of exposures (which cause no stall) at the expense of the number of validations. Finally, InvisiSpec does create more cache hierarchy traffic and contention to various cache ports. The hardware should be designed to handle them.

C. When to Use Validation and Exposure

The memory consistency model determines when to use a validation and when to use an exposure. Consider TSO first. In a high-performance TSO implementation, a speculative load that reads when there is no older load (or fence) in the ROB will not be squashed by a subsequent incoming invalidation to the line it read. Hence, such a USL can use an exposure when it becomes visible. On the other hand, a speculative load that reads when there is at least one older load (or fence) in the ROB will be squashed by an invalidation to the line it read. Hence, such a USL is required to use a validation.

Now consider RC. In this case, only speculative loads that read when there is at least one earlier fence in the ROB will be squashed by an invalidation to the line read. Hence, only those will be required to use a validation; the very large majority of loads can use exposures.

From this discussion, we see that the design with the highest chance of observing validations that stall the pipeline is IS-Future under TSO. To reduce the chance of these events, InvisiSpec implements two mechanisms. The first one enables some USLs that would use validations to use exposures instead. The second one identifies USLs that would use validations, and squashes them early if there is a high chance that their validations would fail. We consider these mechanisms next.

1) Transforming a Validation USL into an Exposure USL: Assume that, under TSO, USL_1 initiates a read while there are earlier loads in the ROB. Ordinarily, InvisiSpec would mark USL_1 as needing a validation. However, assume that, at the time of the read, all of the loads in the ROB earlier than USL_1 have already obtained the data they requested—in particular, if they are USLs, the data they requested has already arrived at the SB and been passed to a register. In this case, USL_1 is not reordered relative to any of its earlier loads. As a result, TSO would not require squashing USL_1 on reception of an invalidation to the line it loaded. Therefore, USL_1 is marked as needing exposure, not validation.

To support this mechanism, we tag each USL with one bit called *Performed*. It is set when the data requested by the USL has been received in the SB and passed to the destination register.

2) Early Squashing of USLs Needing Validations: Assume that a core receives an invalidation for a line in its cache that also happens to be loaded into its SB by a USL marked as needing validation. Receiving an invalidation indicates that the line has been updated. Such update will typically cause the validation of the USL at the point of visibility to fail. Validation could only succeed if this invalidation was caused by false sharing, or if the net effect of all the updates to the line until the validation turned out to be silent (i.e., they restored the data to its initial value). Since these conditions are unlikely, InvisiSpec squashes such a USL on reception of the invalidation.

There is a second case of a USL with a high chance of validation failure. Assume that USL_1 needs validation and has data in the SB. Moreover, there is an earlier USL_2 to the same line (but to different words of the line) that also has its data in the SB and needs validation. When USL_2 performs the validation and brings the line to the core, InvisiSpec also compares the line to USL_1's data in the SB. If the data are different, it shows that USL_1 has read stale data. At that point, InvisiSpec conservatively squashes USL_1.

D. Overlapping Validations and Exposures

To improve performance, we seek to overlap validation and exposure transactions as much as possible. To ensure that overlaps are legal, we propose two requirements. The first one is related to correctness: during the overlapped execution of multiple validation and exposure transactions, the memory consistency model has to be enforced. This is the case even if some of the loads involved are squashed and restarted.

The second requirement only applies to the Futuristic attack model, and is related to the cache state: during the overlapped execution of multiple validation and exposure transactions, no squashed transaction can change the state of the caches. This is because the essence of our Futuristic model defense is to prevent squashed loads from changing the state of the caches. For the Spectre attack model, this requirement does not apply because validations and exposures are only issued for instructions in the correct path of a branch. While some of these instructions may get squashed (e.g., due to memory consistency violations) and change the state of the caches, these are correct program instructions and, hence, pose no threat under the Spectre attack model

We propose sufficient conditions to ensure the two requirements. First, to enforce the correctness requirement, we require that the validation and exposure transactions of the different USLs start in program order. This condition, plus the fact that no validation or exposure for a USL can start until that USL has already received the response to its initial speculative request, ensures that the memory consistency model is enforced. More details are in the Appendix.

To enforce the second requirement for the Futuristic model, we require the following conditions. First, if a USL issues a

validation transaction, no subsequent validation or exposure transaction can overlap with it. This is because, if the validation fails, it will squash all subsequent loads. Second, if a USL issues an exposure transaction, all subsequent exposure transactions up to, and including, the next validation transaction can overlap with it. This is allowed because exposures never cause squashes. In the Spectre model defense, validations and exposures can all overlap—since the cache state requirement does not apply.

Recall that validations and exposures bring cache lines. Hence, validations and/or exposures to the same cache line have to be totally ordered.

E. Reusing the Data in the Speculative Buffer

To exploit the spatial locality of a core's speculative loads, a USL brings a full cache line into the SB, and sets an Address Mask that identifies which part of the line is passed to a register. If a second USL that is *later* in program order now requests data from the line that the first USL has already requested or even brought into the SB, the second USL does not issue a second cache hierarchy access. Instead, it waits until the first USL brings the line into the first USL's SB entry. Then, it copies the line into its own SB entry, and sets its Address Mask accordingly.

F. Reducing Main-Memory Accesses

A USL performs two transactions—one when it is first issued, and one when it validates or exposes. Now, suppose that a USL's first access misses in the last level cache (LLC) and accesses main memory. Then, it is likely that the USL's second access is also a long-latency access to main memory.

To improve performance, we design InvisiSpec to avoid this second main-memory access most of the time. Specifically, we add a per-core LLC Speculative Buffer (LLC-SB) next to the LLC. When a USL's first access reads the line from main memory, as the line is sent back to the requesting core's L1 SB, InvisiSpec stores a copy of it in the core's LLC-SB. Later, when the USL issues its validation or exposure, it will read the line from the core's LLC-SB, skipping the access to main memory.

If, in between the two accesses, a second core accesses the line with a validation/exposure or a safe access, InvisiSpec invalidates the line from the first core's LLC-SB. This conservatively ensures that the LLC-SB does not hold stale data. In this case, the validation or exposure transaction of the original USL will obtain the latest copy of the line from wherever it is in the cache hierarchy.

VI. Detailed InvisiSpec Design

A. Speculative Buffer in L1

1) Speculative Buffer Design: InvisiSpec places the SB close to the core to keep the access latency low. Our main goal in designing the SB is to keep its operation simple, rather then minimizing its area; more area-efficient designs can be developed. Hence, we design the SB with as many entries as the Load Queue (LQ), and a one-to-one mapping between the LQ and SB entries (Figure 3).

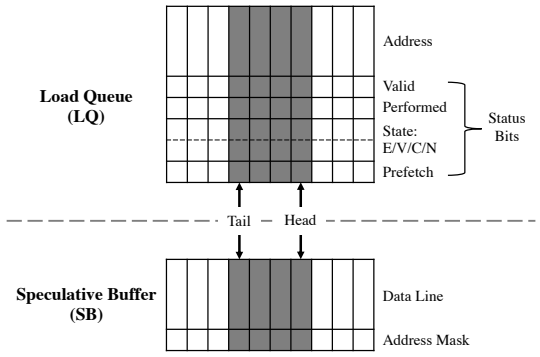

Fig. 3: Speculative Buffer and its logical connection to the load queue.

This design makes several operations easy to support. Given an LQ entry, InvisiSpec can quickly find its corresponding SB entry. Further, since the LQ can easily find if there are multiple accesses to the same address, InvisiSpec can also identify multiple SB entries for the same line. Importantly, it is trivial to (i) allocate an SB entry for the next load in program order, (ii) remove the SB entry for a retiring load at the ROB head, and (iii) remove the SB entries for a set of loads being squashed. These operations need simple moves of SB's Head and Tail pointers—which are the LQ's Head and Tail pointers.

An SB entry does not store any address. It stores the data of a cache line plus an Address Mask that indicates which bytes were read. Each LQ entry has some status bits: Valid, Performed, State, and Prefetch. *Valid* records whether the entry is valid. *Performed* indicates whether the data requested by the USL has arrived and is stored in the SB entry. Recall that this information is used to change a validation to a cheaper exposure (Section V-C1). *State* indicates the state of the load. It can be "requiring an exposure when it becomes visible" (E), "requiring a validation when it becomes visible" (V), "exposure or validation has completed" (C), and "invisible speculation is not necessary for this load" (N). The latter is used when invisible speculation is not needed, and the access should go directly to the cache hierarchy. Finally, *Prefetch* indicates whether this entry corresponds to a prefetch (Section VI-B).

2) Operation of the Load Queue and Speculative Buffer: We describe the LQ and SB algorithms at key events.

A load instruction is issued: The hardware allocates an LQ entry and an SB entry. The LQ entry's Valid bit is set.

The address of a load is resolved: The load is ready to be sent to the cache hierarchy. If the load is safe according to the attack model, the State bits in the LQ entry are set to N and the load is issued to the cache hierarchy with a normal coherence transaction. The SB entry will be unused.

Otherwise, the load is a USL, and the State is set to E or V, as dictated by the memory consistency model (Section V-C). The USL is issued to the cache hierarchy with a new *Spec-GetS* transaction. This transaction requests the line in an invisible mode, without changing any state in any cache. The address mask in the SB entry is set. More details on how *Spec-GetS* changes the coherence protocol are given in Section VI-E1.

Similar to conventional core designs, as a *Spec-GetS* request is issued, the hardware first tries to reuse any relevant local state.

However, InvisiSpec reuses only state allocated by prior (in program order) instructions, to avoid creating new side channels (Section VII). Specifically, the LQ determines whether there is any prior load in the LQ that already requested the same line. If so, and if the line has been received by the SB, InvisiSpec obtains the line from the SB entry where it is, and copies it to the requesting USL's SB entry. If, instead, the line has not yet been received, the requesting USL modifies the prior load's MSHR waiting for the line, so that the line is also delivered to its own SB entry on arrival. If there is a pending request for the line by a later (in program order) load, the USL issues a *Spec-GetS* that allocates a new MSHR. Having multiple *Spec-GetS* in flight for the same address does not pose a problem because they do not change directory state (Section VI-E1).

Finally, if there is a prior store in the ROB to the same address, the store data is forwarded to the USL's destination register, and to the USL's SB entry. The Address Mask of the SB entry is set. Then, a *Spec-GetS* transaction is still issued to the cache hierarchy, while recording that, when the line is received, it should not overwrite the bytes in the SB entry that are marked by the Address Mask. We choose this design to treat all SB entries equally: they should contain the data line that their USLs have read speculatively.

The line requested by a USL arrives at the core: The line is copied to the SB entry of the requesting USL, and the requested bytes are passed to the USL's destination register—unless, as indicated above, the data has already been forwarded by a store. The Performed bit of the LQ entry is set. All these operations are performed for the potentially multiple USLs waiting for this line. These operations are performed no matter where the line comes from; in particular, it could come from the local L1 cache, which remains unchanged.

A USL reaches the visibility point: If the USL's State in the LQ is V, InvisiSpec issues a validation; if it is E, InvisiSpec issues an exposure. These transactions can also reuse MSHRs of earlier requests (in program order) to the same line.

The response to an exposure or validation arrives: The incoming line is saved in the local cache hierarchy as in regular transactions. If this was an exposure response, the requesting USL may or may not have retired. If it has not, InvisiSpec sets the USL's State to C. If this was a validation response, the USL has not retired and needs to be validated. The validation proceeds by comparing the bytes that were read by the USL with the same bytes in the incoming cache line. If they match, the USL's State is set to C; otherwise, the USL is squashed—together with all the subsequent instructions. Squashing moves the LQ and SB tail pointer forward.

All these operations are performed for the potentially multiple USLs waiting for this line to become visible. At this point, InvisiSpec also can perform the early squash operation of Section V-C2.

Note that, in IS-Spectre, squashing a USL can lead to squashing subsequent USLs that have already become visible. Squashing such USLs does not cause any security problem, because they are safe according to IS-Spectre, and so their microarchitectural side-effects can be observed.

An incoming cache line invalidation is received: In general, this event does not affect the lines in the SB; such lines are invisible to the cache coherence protocol. However, we implement the optimization of Section V-C2, where some USLs may be conservatively squashed. Specifically, the LQ is searched for any USL with the Performed bit set that has read from the line being invalidated, and that requires validation (i.e., its State is V). These USLs are conservatively squashed, together with their subsequent instructions. Of course, as in conventional cores, the incoming invalidation may affect other loads in the LQ that have brought lines into the caches.

A cache line is evicted from the local L1: Since the lines in the SB are invisible to the cache coherence protocol, they are unaffected. As in conventional cores, the eviction may affect other loads in the LQ that have brought lines into the caches.

3) Primitive Operations: Table III shows the primitive operations of the SB. Thanks to the SB design, all the operations are simple. Only the comparison of data in a validation is in the critical path.

Operation	How It is Done	Complexity?	Critical Path?
Insert the data line requested by a USL	Index the SB with the same index as the LQ. Fill the entry	Low	No
Validate an SB entry	Use the Address Mask to compare the data in the SB entry to the incoming data	Low	Yes
Copy one SB entry to another	Read the data from one entry and write it to another	Low	No

TABLE III: Primitive operations of the SB.

B. Supporting Prefetching

InvisiSpec supports software prefetch instructions. Such instructions follow the same two steps as a USL. The first step is an "invisible" prefetch that brings a line to the SB without changing any cache hierarchy state, and allows subsequent USLs to access the data locally. The second one is an ordinary prefetch that brings the line to the cache when the prefetch can be made visible. This second access is an exposure, since prefetches need not go through memory consistency checks.

To support software prefetches, InvisiSpec increases the size of a core's SB and, consequently, LQ. The new size of each of these structures is equal to the maximum number of loads and prefetches that can be supported by the core at any given time. InvisiSpec marks the prefetch entries in the LQ with a set *Prefetch* bit.

To be secure, InvisiSpec does not support speculative hardware prefetching. Only when a load or another instruction is made visible, can that instruction trigger a hardware prefetch.

C. Per-Core Speculative Buffer in the LLC

InvisiSpec adds a per-core LLC-SB next to the LLC. Its purpose is to store lines that USLs from the owner core have requested from main memory, and to provide the lines when InvisiSpec issues the validations or exposures for the same loads—hence avoiding a second access to main memory.

To understand the LLC-SB design, consider the case of a USL that issues a request that will miss in the LLC and access the LLC-SB. However, before the USL receives the data, the

USL gets squashed, is re-issued, and re-sends the request to the LLC-SB. First, for security reasons, we do not want the second request to use data loaded in the LLC-SB by a prior, squashed request (Section VII). Hence, InvisiSpec forbids a USL from obtaining data from the LLC-SB; if a USL request misses in the LLC, the request bypasses the LLC-SB and accesses main memory. Second, it is possible that the two USL requests get reordered on their way to the LLC-SB, which would confuse the LLC-SB. Hence, we add an Epoch ID to each core, which is a counter that the hardware increments every time the core squashes instructions. When a core communicates with its LLC-SB, it includes its Epoch ID in the message. With this support, even if two USL requests from different epochs are reordered in transit, the LLC-SB will know that, given two requests with different IDs, the one with the higher ID is the correct one.

We propose a simple design of the per-core LLC-SB, which can be easily optimized. It is a circular buffer with as many entries as the LQ, and a one-to-one mapping between LQ and LLC-SB entries. Each LLC-SB entry stores the data of a line, its address, and the ID of the epoch when the line was loaded. USL requests and validation/exposure messages contain the address requested, the index of the LLC-SB where the data should be written to or read from, and the current Epoch ID. With this design, the algorithm works as follows:

A USL request misses in the LLC: The request skips the LLC-SB and reads the line from memory. Before saving the line in the indexed LLC-SB entry, it checks the entry's Epoch ID. If it is higher than the request's own Epoch ID, the request is stale and is dropped. Otherwise, line, address, and Epoch ID are saved in the entry, and the line is sent to the core.

A validation/exposure request misses in the LLC: The request checks the indexed LLC-SB entry. If the address and Epoch ID match, InvisiSpec returns the line to the core, therefore saving a main memory access. Otherwise, InvisiSpec accesses main memory and returns the data there to the core. In both cases, InvisiSpec writes the line into the LLC, and invalidates the line from the LLC-SBs of all the cores (including the requesting core). This step is required to purge future potentially-stale data from the LLC-SBs (Section V-F). Invalidating the line from the LLC-SBs requires search, but is not in the critical path, as InvisiSpec does it in the background, as the line is being read from main memory and/or delivered to the requesting core.

A safe load misses in the LLC: The request skips the LLC-SB and gets the line from main memory. In the shadow of the LLC miss, InvisiSpec invalidates the line from the LLC-SBs of all the cores, as in the previous case. The line is loaded into the LLC.

D. Disabling Interrupts

In IS-Future, the hardware can initiate a validation or exposure only when the USL becomes either (i) non-speculative because it reaches the ROB head or (ii) speculative non-squashable by any earlier instruction. If we wait until the load reaches the ROB head to start the validation, the pipeline may stall. Therefore, it is best to initiate the validation as soon as the load becomes speculative non-squashable. As indicated in Section V-A1, speculative non-squashable loads are loads that, while not at the ROB head, are preceded in the ROB only by instructions that cannot be squashed by any of the squashing events in Table I for Futuristic. As shown in the table, one of the squashing events is interrupts. Therefore, to make a load speculative non-squashable, interrupts need to be delayed from the time that the load would otherwise be speculative non-squashable, until the time that the load reaches the ROB head.

To satisfy this condition, InvisiSpec has the ability to automatically, transparently, and in hardware disable interrupts for very short periods of time. Specifically, given a USL, when the hardware notices that none of the instructions earlier than the USL in the ROB can be squashed by any of the squashing events in Table I for Futuristic except for interrupts, the hardware disables interrupts. The validation or exposure of the USL can then be initiatiated. As soon as the USL reaches the head of the ROB, interrupts are automatically enabled again. They remain enabled for at least a minimum period of time, to ensure that interrupts are not starved.

E. Other Implementation Aspects

1) Changes to the Coherence Protocol: InvisiSpec adds a new *Spec-GetS* coherence transaction, which obtains a copy of the latest version of a cache line from the cache hierarchy without changing any cache or coherence states. For instance, in a directory protocol, a *Spec-GetS* obtains the line from the directory if the directory owns it; otherwise, the directory forwards the request to the owner, which sends a copy of the line to the requester. The directory does not order forwarded *Spec-GetS* requests with respect to other coherence transactions, to avoid making any state changes. For this reason, if a forwarded *Spec-GetS* arrives at a core after the core has lost ownership of the line, the *Spec-GetS* is bounced back to the requester, which retries. The requesting USL cannot starve as a result of such bounces, because eventually it either gets squashed or becomes safe. In the latter case, it then issues a standard coherence transaction.

2) Atomic Instructions: Since an atomic instruction involves a write, InvisiSpec does not execute them speculatively. Execution is delayed as in current processors.

3) Securing the D-TLB: To prevent a USL from observably changing the D-TLB state, InvisiSpec uses a simple approach. First, on a D-TLB miss, it delays serving it via a page table walk until the USL reaches the point of visibility. If the USL is squashed prior to that point, no page table walk is performed. Second, on a D-TLB hit, any observable TLB state changes such as updating D-TLB replacement state or access/dirty bits are delayed to the USL's point of visibility. A more sophisticated approach would involve using an SB structure like the one used for the caches.

4) No ABA Issues: In an ABA scenario, a USL reads value A into the SB, and then the memory location changes to B and back to A prior to the USL's validation. An ABA scenario does not violate the memory model in an InvisiSpec validation.

VII. Security Analysis

InvisiSpec's SB and LLC-SB do not create new side channels. To see why, we consider the ways in which a transient, destined to be squashed USL (i.e., the *transmitter*) could try to speed up or slow down the execution of a load that later retires (i.e., the *receiver*). We note that if this second load is also squashed, there is no security concern, since a squashed load does not have side effects, and hence does not pose a threat.

Speeding up. The transmitter could attempt to speed up the receiver's execution by accessing the same line as the receiver, so that the latter would get its data with low latency from the SB or LLC-SB entry allocated by the transmitter. For this to happen, the transmitter has to execute before the receiver. In the following, we show that this side channel cannot happen. To see why, consider the two possible cases:

a) The transmitter comes before the receiver in program order: In this case, the receiver has to be issued after the transmitter is actually squashed. Otherwise, the receiver would be squashed at the same time as the transmitter is. However, when the transmitter is squashed, its entries in the SB and LLC-SB become unusable by a later request: the SB entry's Valid bit is reset, and the LLC-SB entry's Epoch ID tag becomes stale, as the receiver gets a higher Epoch ID. As a result, the receiver cannot reuse any state left behind by the transmitter.

b) The transmitter comes after the receiver in program order: This is the case when, because of out-of-order execution, the transmitter has already requested the line (and maybe even loaded it into its own SB or LLC-SB entries) by the time the receiver requests the line. The receiver could be sped-up only if it could leverage the transmitter's earlier request or buffered data. However, InvisiSpec does not allow a load (USL or otherwise) to reuse any state (e.g., state in an SB entry or in an MHSR entry) allocated by a USL that is later in program order. Instead, the receiver issues its own request to the cache hierarchy (Section VI-A2) and is unaffected by the transmitter. For the LLC-SB, the only state reuse allowed occurs when a validation/exposure for a load reuses the entry left by the *Spec-GetS* request of the same load.

Slowing down. The transmitter could attempt to slow down the receiver's execution by allocating all the entries in one of the buffers. But recall that the receiver must retire for the slow-down to be observable. Therefore, the receiver must come before the transmitter in program order. However, SB and LLC-SB entries are allocated at issue time, due to their correspondence to LQ entries. Therefore, allocation of SB or LLC-SB entries by the later transmitter cannot affect the allocation ability of the earlier receiver. Finally, contention on other resources, such as MSHRs or execution units, could slow down the receiver, but such side channels are considered out of scope in this paper (Section III-B).

VIII. Experimental Setup

To evaluate InvisiSpec, we modify the Gem5 [33] simulator, which is a cycle-level simulator with support for modeling the side effects of squashed instructions. We run individual SPECInt2006 and SPECFP2006 applications [34] on a single core, and multi-threaded PARSEC applications [35] on 8 cores. For SPEC, we use the *reference* input size and skip the first 10 billion instructions; then, we simulate for 1 billion instructions. For PARSEC we use the *simmedium* input size and simulate the whole region-of-interest (ROI). Table IV shows the parameters of the simulated architecture. When running a SPEC application, we only enable one bank of the shared cache.

Parameter	Value
Architecture	1 core (SPEC) or 8 cores (PARSEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries, Tournament branch predictor, 4096 BTB entries, 16 RAS entries
Private L1-I Cache	32KB, 64B line, 4-way, 1 cycle round-trip (RT) lat., 1 port
Private L1-D Cache	64KB, 64B line, 8-way, 1 cycle RT latency, 3 Rd/Wr ports
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 8 cycles RT local latency, 16 cycles RT remote latency (max)
Network	4×2 mesh, 128b link width, 1 cycle latency per hop
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50 ns after L2

TABLE IV: Parameters of the simulated architecture.

We model the 5 processor configurations shown in Table V: *Base* is a conventional, insecure processor, *Fe-Sp* inserts a fence after every indirect/conditional branch, *IS-Sp* is InvisiSpec-Spectre, *Fe-Fu* inserts a fence before every load, and *IS-Fu* is InvisiSpec-Future. We model both TSO and RC.

Names		Configurations
Base	UnsafeBaseline	Conventional, insecure baseline processor
Fe-Sp	Fence-Spectre	Insert a fence after every indirect/conditional branch
IS-Sp	InvisiSpec-Spectre	USL modifies only SB, and is made visible after all the preceding branches are resolved
Fe-Fu	Fence-Future	Insert a fence before every load instruction
IS-Fu	InvisiSpec-Future	USL modifies only SB, and is made visible when it is either non-speculative or spec non-squashable

TABLE V: Simulated processor configurations.

In InvisiSpec-Future, a USL is ready to initiate a validation/-exposure when the following is true for all the instructions before the USL in the ROB: (i) they cannot suffer exceptions anymore, (ii) there are no unresolved control-flow instructions, (iii) all stores have retired into the write buffer, (iv) all loads have either finished their validation or initiated their exposure transaction, and (v) all synchronization and fence instructions have completed. At that point, we temporarily disable interrupts and initiate the validation/exposure. These are slightly more conservative conditions than those listed in Table I for Futuristic.

IX. Evaluation

A. Proof-of-Concept Defense Analysis

We evaluate the effectiveness of InvisiSpec-Spectre at defending against the attack in Figure 1. We set the secret value V to 84. After triggering the misprediction in the victim, the attacker scans array B and reports the access latency. Figure 5 shows the median access latency for each cache line measured by the attacker after 100 times.

From the figure, we see that under *Base*, the attacker can obtain the secret value. Only the access to the line corresponding to the secret value hits in the caches, and takes less than 40 cycles. All the other accesses go to main memory,

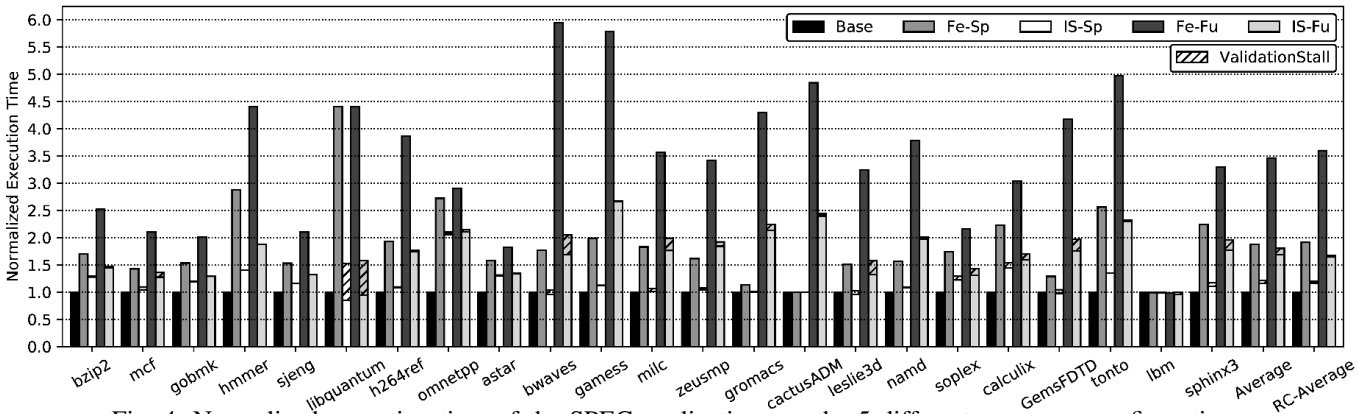

Fig. 4: Normalized execution time of the SPEC applications on the 5 different processor configurations.

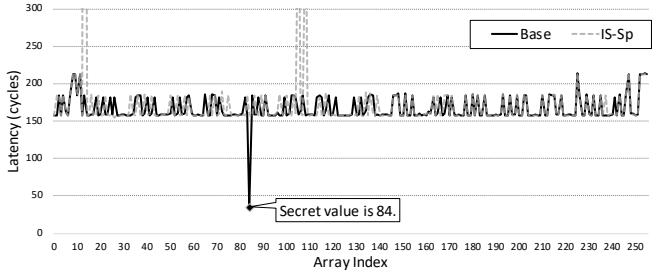

Fig. 5: Access latency measured in the code of Figure 1.

and take over 150 cycles. However, with *IS-Sp*, the attack is successfully thwarted. All the accesses to all the lines go to main memory because loads in the mispredicted path of a branch do not change the cache state, and the SB does not leak information from squashed loads as discussed in Section VII.

B. Performance Evaluation of the SPEC Applications

Execution time. Figure 4 compares the execution time of the SPEC applications on the 5 processor configurations of Table V. From left to right, we show data for each application under TSO, for the average application under TSO, and for the average under RC (*RC-Average*). Each set of bars is normalized to *Base*. For *IS-Sp* and *IS-Fu*, we show the contribution of the stall caused by validation operations.

If we focus on the fence-based solutions, we see that they have high overhead. Under TSO, the average execution time of *Fe-Sp* and *Fe-Fu* is 88% and 246% higher, respectively, than *Base*. The overhead of InvisiSpec is lower, although still significant. Under TSO, the average execution time of *IS-Sp* and *IS-Fu* is 22% and 80% higher, respectively, than *Base*.

There are three main reasons for the slowdowns of *IS-Sp* and *IS-Fu*: validation stalls, lack of cache state reuse after branch squash, and contention due to two accesses per load. We consider each of them in turn.

The figure shows that, for most applications, validation stalls in *IS-Sp* and *IS-Fu* are minor. The reason is that most of the validations hit in the L1 cache and are served quickly. Memory-bound applications with high LLC miss rates, such as *libquantum* and *bwaves*, suffer the most from validations. In these cases, validation hits in the LLC-SB reduce the overhead.

In *IS-Sp*, the application with the highest execution overhead is *omnetpp* (around 100%). The main reason is the lack of cache state reuse after branch squash. *omnetpp* has both a high branch misprediction rate and a high LLC miss rate. In *Base*, instructions in a branch path that will be squashed still bring useful data into the cache, which then speeds-up the execution of instructions in the correct branch path. This is common in branches where both directions quickly converge to the same instruction flow. In contrast, *IS-Sp* and *IS-Fu* do not allow squashed loads to modify the cache.

In *IS-Fu*, applications with a high rate of memory system accesses such as *gamess*, *cactusADM*, and *tonto* have the highest execution time increase (around 100-150%). About 30% of the instructions in these applications are load instructions. The resulting high rate of USLs, many inducing two cache hierarchy accesses, causes contention in the cache hierarchy and slows down execution.

In RC, the average execution time in *IS-Sp* and *IS-Fu* is 20% and 67% higher than in *Base*. This is a bit less than TSO.

Network traffic. We record the network traffic, measured as the total number of bytes transmitted between caches, and between cache and main memory. Figure 6 compares the traffic in the 5 processor configurations of Table V. The bars are organized as in Figure 4. For *IS-Sp* and *IS-Fu*, we show the fraction of the traffic induced by USLs (*SpecLoad*) and exposures/validations. The rest of the bar is due to non-speculative accesses.

Under TSO, the average network traffic in *IS-Sp* and *IS-Fu* is 34% and 60% higher, respectively, than *Base*. The traffic increase is about the same under RC.

On average, the traffic in *IS-Sp* and *IS-Fu* without the exposures/validations is a bit higher than in *Base*. On top of that, we have the exposure and validation traffic. However, in some applications—most notably *sjeng*—the traffic without exposures/validations is already much higher than *Base*. The reason is that these programs have a high branch misprediction rate. This leads to many USLs that get squashed, increasing the *SpecLoad* category. Note that a given load may be squashed multiple times in a row, if it is in the shadow of multiple mispredicted branches—inducing a large *SpecLoad* category. In programs with a large *SpecLoad* category, the *Expose/Validate* category is usually small. The reason is that, by the time a USL is squashed for its last time, it may be reissued as a non-speculative load, and not require exposure/validation.

On average, the traffic in *Fe-Sp* and *Fe-Fu* is like in *Base*.

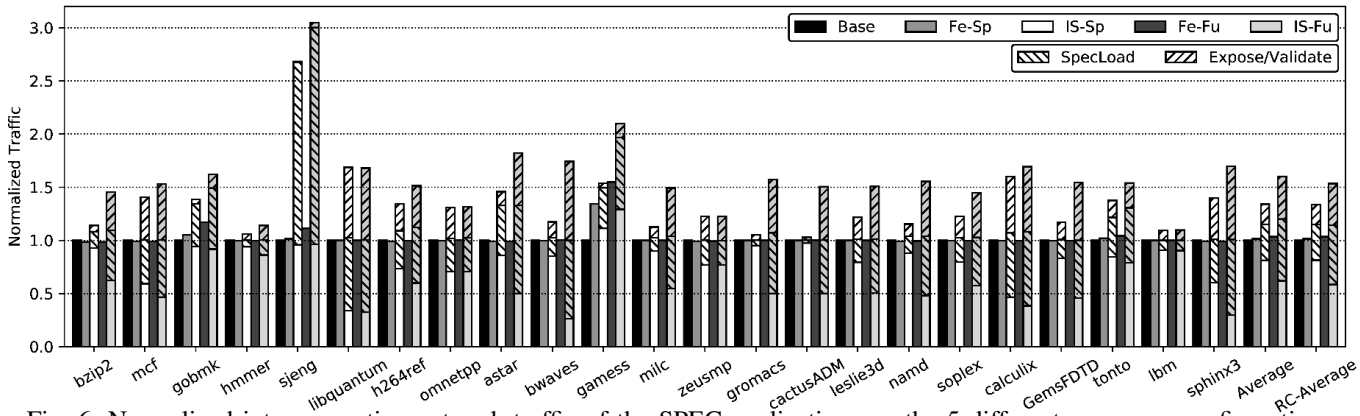

Fig. 6: Normalized interconnection network traffic of the SPEC applications on the 5 different processor configurations.

Intuitively, it should be lower than *Base* because *Fe-Sp* and *Fe-Fu* do not execute speculative instructions. In reality, it can be shown that, while the data traffic is lower, the instruction traffic is higher, and the two effects cancel out. The reason is that our simulation system still *fetches* speculative instructions (since this paper only focuses on the data cache hierarchy), without executing them. Fences in the pipeline cause a larger volume of instructions fetched in mispredicted branch paths.

C. Performance Evaluation of the PARSEC Applications

Execution time. Figure 7 shows the normalized execution time of the PARSEC applications on our 5 processor configurations. The figure is organized as in Figure 4. We see that *Fe-Sp* and *Fe-Fu* increase the average execution time over *Base* by a substantial 61% and 171%, respectively. On the other hand, *IS-Sp* and *IS-Fu* increase the execution time by 20% and 65%, respectively. Similar numbers are attained with RC. These slowdowns are smaller than in SPEC programs.

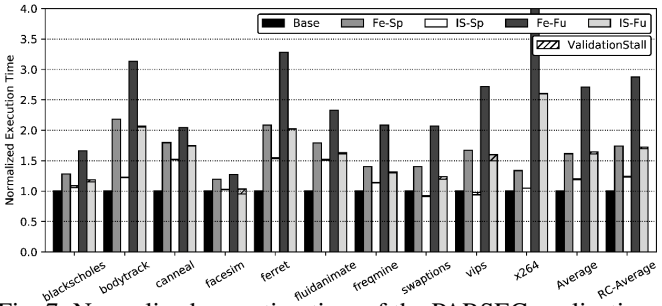

Fig. 7: Normalized execution time of the PARSEC applications.

From Figure 7, we see that validation stalls in *IS-Sp* and *IS-Fu* are minimal. This is because many of the validations are satisfied by the L1 cache. The main reasons for the slowdowns are the lack of cache state reuse after branch squashes, and the resource contention caused by two accesses per speculative load. We consider the higher number of accesses next.

Network traffic. Figure 8 shows the normalized traffic of the PARSEC applications in our 5 configurations. The figure is organized as usual. The *IS-Sp* and *IS-Fu* bars are broken down into traffic induced by USLs (*SpecLoad*), by exposures/validations, and (the rest of the bar) by non-speculative accesses.

On average, *Fe-Sp* and *Fe-Fu* have slightly less traffic than *Base*. The main reason is that they do not execute speculative

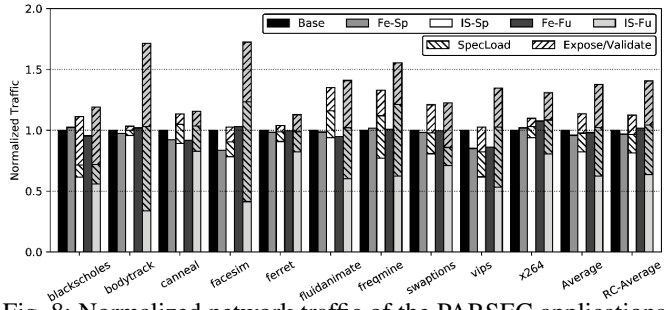

Fig. 8: Normalized network traffic of the PARSEC applications.

instructions. Under TSO, *IS-Sp* and *IS-Fu* increase the traffic by an average of 14% and 38%, respectively, over *Base*. Similar numbers are obtained under RC.

Generally, *IS-Sp* exhibits modest traffic increases. This is because these applications have relatively low branch misprediction rates. *IS-Fu* has more traffic. A good fraction of the traffic typically comes from validations and exposures.

D. Characterization of InvisiSpec's Operation

Table VI shows some statistics related to InvisiSpec's operation under TSO. It shows data on 3 SPEC applications, the average of SPEC, 3 PARSEC applications, and the average of PARSEC. Columns 3-8 break down the total number of exposures and validations into exposures, validations that hit in the L1 cache, and validations that miss in the L1 cache. We see that a good fraction of the transactions are exposures (e.g., 37% in SPEC and 31% in PARSEC for *IS-Fu*). Further, only a small fraction are validations that miss in L1 (e.g., 12% in SPEC and 3% in PARSEC for *IS-Fu*). That is why most applications do not suffer from long validation stalls.

Columns 9-10 show the number of squashes per $1M$ instructions. PARSEC applications have a lower squash rate than SPEC applications. Columns 11-16 break these events into the reason for the squash: branch misprediction, consistency violation, and validation failure. The large majority are caused by branch mispredictions (e.g., 99% in SPEC and 98% in PARSEC for *IS-Fu*), and only a few by consistency violations (e.g., 1% in SPEC and 2% in PARSEC for *IS-Fu*). There are practically no validation failures.

Columns 17-20 show the hit rates in the L1-SB, and in the LLC-SB. The LLC-SB hit rates only apply to validations and

	Application Name	Exposures and Validations						# Squashes Per 1M Instructions		Reason for Squash (%)						L1 SB Hit Rate (%)		LLC-SB Hit Rate in Validations and Exposures (%)	
		% Exposures		% L1 Hit Validations		% L1 Miss Validations				Branch Misprediction		Consistency Violation		Validation Failure					
		Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu	Sp	Fu
SPEC	sjeng	34.2	49.0	64.7	51.0	1.1	0.0	75449	79105	100.0	100.0	0.0	0.0	0.0	0.0	0.0	0.0	89.5	89.7
	libquantum	0.9	1.0	0.0	0.0	99.1	99.0	0	0	100.0	100.0	0.0	0.0	0.0	0.0	3.1	3.8	100.0	100.0
	omnetpp	40.6	42.6	50.7	49.5	8.7	7.9	17001	17667	100.0	100.0	0.0	0.0	0.0	0.0	4.8	4.5	99.9	99.8
	Average	18.4	37.1	63.9	51.0	17.7	11.9	14734	15277	97.5	99.1	2.5	0.9	0.0	0.0	0.8	1.3	96.2	97.5
PARSEC	bodytrack	8.5	36.2	88.9	62.7	2.6	1.1	2439	2487	93.2	92.3	6.8	7.7	0.0	0.0	0.2	0.2	97.9	98.1
	fluidanimate	23.7	31.7	74.8	66.8	1.5	1.5	4291	4338	100.0	100.0	0.0	0.0	0.0	0.0	0.1	0.3	85.2	96.4
	swaptions	15.9	31.2	71.2	67.4	12.9	1.4	2316	2350	97.1	96.8	2.9	3.2	0.0	0.0	0.3	0.1	96.2	96.0
	Average	19.3	31.1	73.3	65.5	7.4	3.4	4074	4296	98.2	97.8	1.8	2.2	0.0	0.0	0.2	0.6	89.7	94.0

TABLE VI: Characterization of InvisiSpec's operation under TSO. Sp and Fu stand for *IS-Sp* and *IS-Fu*.

exposures. On average, L1-SB hit rates are very low (1.3% in SPEC and 0.6% in PARSEC for *IS-Fu*), while LLC-SB hit rates are very high (98% in SPEC and 94% in PARSEC for *IS-Fu*). High LLC-SB hit rates boost performance.

To give an idea of the range of values observed, the table also shows data for a few individual applications. Finally, we collected the same statistics under RC. Under RC, there are practically no validations (i.e., practically all are exposures). Further, there are very few consistency violations, and practically all squashes are due to branch mispredictions.

E. Estimating Hardware Overhead

InvisiSpec adds two main per-core structures, namely, the L1-SB (a cache) and the LLC-SB (a CAM). We use CACTI 5 [36] to estimate, at 16nm, their area, access time, dynamic read and write energies, and leakage power. These estimates, shown in Table VII, do not include their random logic. Overall, these structures add modest overhead.

Metric	L1-SB	LLC-SB
Area (mm2)	0.0174	0.0176
Access time (ps)	97.1	97.1
Dynamic read energy (pJ)	4.4	4.4
Dynamic write energy (pJ)	4.3	4.3
Leakage power (mW)	0.56	0.61

TABLE VII: Per-core hardware overhead of InvisiSpec.

X. Related Work

Concurrent to our work, SafeSpec [37] proposes to defend against speculative execution attacks via a shadow structure that holds speculative state for caches and TLBs. This structure is similar to our SB in InvisiSpec. However, SafeSpec does not handle cache coherence or memory consistency model issues and, therefore, cannot support multithreaded workloads. Renau [38] provides a classification of side channel leaks that result from different predictors present in typical high-performance cores. He also lists several high-level ideas on how to protect from speculative time leaks, such as avoiding or fixing speculative updates. InvisiSpec is different in that it is a concrete and highly-detailed solution. There have been other proposals to defend against speculative execution attacks [39], [6], [40]. As discussed in Section I, they all have performance, usability, or completeness issues. Further, none of the schemes considers Futuristic attacks.

Existing defense mechanisms against cache-based side channel attacks are insufficient to defeat speculative execution attacks. Cache partition techniques [41], [42], [43] work in cross-core or SMT settings, but cannot deal with the same-thread setting (Section III), because they only block cache interference between different processes or security domains. Other mechanisms, such as Catalyst [44] and StealthMem [45], prevent an attacker from observing a victim's access patterns on a secure-sensitive region. Such protections are likely ineffective in speculative execution attacks, where accesses are unlikely to be within the selected security-sensitive region—e.g., out-of-bounds array accesses.

Martinez et al. [46] and Bell and Lipasti [30] identify conditions for an instruction to retire early. We use their ideas. In addition, Cain and Lipasti [31] propose to enforce memory consistency by re-executing loads prior to retirement. This is somewhat similar to InvisiSpec's validation technique. However, in their design, there is no equivalent to an exposure. Further, all of their loads modify the cache.

XI. Conclusion and Future Work

This paper presented InvisiSpec, a novel approach to defend against hardware speculation attacks in multiprocessors by making speculation invisible in the data cache hierarchy. In InvisiSpec, unsafe speculative loads read data into a speculative buffer, without modifying the cache hierarchy. When the loads are safe, they are made visible to the rest of the system through validations or exposures. We proposed an InvisiSpec design to defend against Spectre-like attacks, and one to defend against futuristic attacks where any speculative load may pose a threat. Our evaluation showed that, under TSO, using fences to defend against Spectre attacks slows down execution by an average of 74% over a conventional, insecure processor; InvisiSpec induces an execution slowdown of only 21%. Using fences to defend against futuristic attacks slows down execution by an average of 208%; InvisiSpec induces a slowdown of 72%.

Our current work involves improving InvisiSpec to reduce its execution overhead. One direction involves leveraging the fact that many loads can be proven safe in advance, and do not require InvisiSpec's hardware. A second one involves redesigning InvisiSpec's mechanisms to be more aggressive.

Acknowledgments

This work was funded in part by NSF under grant CCF-1725734. Adam Morrison is supported in part by Len Blavatnik and the Blavatnik Family Foundation.

Appendix: Ordering of Validations and Exposures

Validation and exposure transactions have to be initiated in program order to guarantee that TSO memory consistency is

maintained. To see why, recall that the only load-load reordering that leads to an observable violation of TSO is one in which two loads from processor P1, which in program order are $ld(y)$ first and $ld(x)$ second, interact with two stores from other processors, $st(x)$ and $st(y)$, such that they end up being globally ordered as: $ld(x) \rightarrow st(x) \rightarrow st(y) \rightarrow ld(y)$ [47]. In other words, $ld(x)$ reads a value that gets overwritten by $st(x)$, $st(y)$ is globally ordered after $st(x)$, and $ld(y)$ reads the value stored by $st(y)$. Assume that the USLs for $ld(y)$ and $ld(x)$ have proceeded out of order and resulted in the above global order. However, in-order initiation of validations/exposures implies that when $ld(x)$'s validation is initiated, the USL for $ld(y)$ has already returned the data. If the latter returned the value stored by $st(y)$, then the validation of $ld(x)$ must now obtain the value stored by $st(x)$. Therefore, $ld(x)$'s validation will fail, and $ld(x)$ will be squashed. This enforces TSO.

RC does not prevent load-load reordering unless the loads are separated by a fence or synchronization. Hence, the in-order validation/exposure initiation requirement can be relaxed. For simplicity, this paper enforces this ordering for RC as well.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *IEEE S&P*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[3] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE S&P*, 2013.

[4] D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES," in *CT-RSA*, 2006.

[5] F. Yao, M. Doroslovack, and G. Venkataramani, "Are Coherence Protocol States Vulnerable to Information Leakage?" in *HPCA*, 2018.

[6] Intel, "Speculative Execution Side Channel Mitigations," https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[7] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Res. and Dev.*, 1967.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach*. Elsevier, 2011.

[9] M. Johnson, *Superscalar Microprocessor Design*. Prentice Hall Englewood Cliffs, New Jersey, 1991.

[10] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Pub., 2011.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy, "Two Techniques to Enhance the Performance of Memory Consistency Models," in *ICPP*, 1991.

[12] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: a Rigorous and Usable Programmer's Model for x86 Multiprocessors," *CACM*, Jul. 2010.

[13] *The SPARC Architecture Manual Version 8*. Prentice Hall, 1992.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors," in *ISCA*, 1990.

[15] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read Arbitrary Memory over Network," *ArXiv e-prints*, 2018.

[16] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *IEEE S&P*, 2015.

[17] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World," in *IEEE S&P*, 2019.

[18] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018.

[19] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *CRYPTO*, 1999.

[20] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic, "EDDIE: EM-Based Detection of Deviations in Program Execution," in *ISCA*, 2017.

[21] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *IEEE S&P*, 2015.

[22] H. M. G. Wassel, Y. Gao, J. K. Oberg, T. Huffmire, R. Kastner, F. T. Chong, and T. Sherwood, "SurfNoC: A Low Latency and Provably Non-interfering Approach to Secure Networks-on-chip," in *ISCA*, 2013.

[23] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing Channel Protection for a Shared Memory Controller," in *HPCA*, 2014.

[24] Intel, "Q3 2018 Speculative Execution Side Channel Update," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018.

[25] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[26] Intel, "Lazy FP State Restore," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00145.html, 2018.

[27] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

[28] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *ArXiv e-prints*, Jul. 2018.

[29] Intel, "Q2 2018 Speculative Execution Side Channel Update," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, 2018.

[30] G. Bell and M. Lipasti, "Deconstructing Commit," in *ISPASS*, 2004.

[31] H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value-based Approach," in *ISCA*, 2004.

[32] Intel, "Intel Analysis of Speculative Execution Side Channels White Paper," https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, 2011.

[34] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, 2006.

[35] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *PACT*, 2008.

[36] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP, Tech. Rep., 2008.

[37] K. N. Khasawneh, E. Mohammadian Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," *ArXiv e-prints*, 2018.

[38] J. Renau, "Securing Processors from Time Side Channels," UC Santa Cruz, Tech. Rep., April 2018.

[39] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[40] P. Turner, "Retpoline: a Software Construct for Preventing Branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[41] J. Liedtke, N. Islam, and T. Jaeger, "Preventing Denial-of-service Attacks on a μ-kernel for WebOSes," in *HotOS*, 1997.

[42] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *DSN-W*, 2011.

[43] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection," in *DAC*, 2016.

[44] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing," in *HPCA*, 2016.

[45] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," in *USENIX Security Symposium*, 2012.

[46] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *MICRO*, 2002.

[47] A. Ros, T. E. Carlson, M. Alipour, and S. Kaxiras, "Non-Speculative Load-Load Reordering in TSO," in *ISCA*, 2017.